MinML: Syntax, Static Semantics, and Dynamic Semantics

Amr Sabry

January 24, 2002

1 Introduction

This note describes the syntax, static semantics, and dynamic semantics of a small functional language. The material is based on Chapters 3-6 of Robert Harper's recent manuscript *Programming Languages: Theory and Practice* [1] but the dynamic semantics has been adapted to be as close as possible to the style used by the Jbook [2] for describing the semantics of Java.

2 Syntax

The syntax is given by the following BNF. In the following n ranges over integers; and both x and f range over identifiers:

Types	t	$::=$	$\texttt{int}$	*integer type*
		$\mid$	$\texttt{bool}$	*boolean type*
		$\mid$	$t \rightarrow t$	*function type*
Operators	o	$::=$	$+ \mid - \mid * \mid / \mid = \mid <$	
Expressions	e	$::=$	n	*integer constants*
		$\mid$	x	*variables*
		$\mid$	$o(e, e)$	*primitive applications*
		$\mid$	$\mathsf{true} \mid \mathsf{false}$	*boolean constants*
		$\mid$	$\textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e$	*conditional expressions*
		$\mid$	$(\textbf{fun}\ t\ f\ (t\ x)\ \{e\})$	*user-defined recursive functions*
		$\mid$	$e(e)$	*function applications*

Here are some examples of programs with their intuitive semantics:

p_1	$=$	5	a trivial program
p_2	$=$	$+(+(1,2), +(3,4))$	another trivial program
p_3	$=$	$+(+(1,2), +(/(1,0),4))$	divides by zero
p_5	$=$	$(\textbf{fun}\ \texttt{int}\ f\ (\texttt{int}\ x)\ \{\textbf{if}\ = (x,0)\ \textbf{then}\ 1\ \textbf{else}\ * (x, f(-(x,1)))\})$	factorial
p_5	$=$	$(\textbf{fun}\ \texttt{int}\ f\ (\texttt{int}\ x)\ \{f(x)\})(0)$	infinite loop
p_6	$=$	$\textbf{if}\ \mathsf{true}\ \textbf{then}\ 1\ \textbf{else}\ (\textbf{fun}\ \texttt{int}\ f\ (\texttt{int}\ x)\ \{f(x)\})(0)$	evaluates to 1
p_7	$=$	$+(\mathsf{true}, 1)$	type error
p_8	$=$	$(\textbf{fun}\ \texttt{int}\ f\ (\texttt{bool}\ x)\ \{x\})$	another type error

3 Static Semantics

The static semantics filters the set of syntactically correct programs to exclude those programs that are not well-typed according to the rules below. The intention (which we formalize and prove in Section 5) is that

$$\frac{}{\Gamma \vdash n : \texttt{int}} \qquad \frac{}{\Gamma \vdash \mathsf{true} : \texttt{bool}} \qquad \frac{}{\Gamma \vdash \mathsf{false} : \texttt{bool}} \qquad \frac{}{\Gamma \vdash x : t}\ \text{if } (x : t) \in \Gamma$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash = (e_1, e_2) : \texttt{bool}} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash < (e_1, e_2) : \texttt{bool}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash o(e_1, e_2) : \texttt{int}}\ \text{if } o \text{ is one of } \{+, -, *, /\}$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : t \qquad \Gamma \vdash e_3 : t}{\Gamma \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : t}$$

$$\frac{\Gamma, x : t_x, f : t_x \to t_r \vdash e : t_r}{\Gamma \vdash (\textbf{fun}\ t_r\ f\ (t_x\ x)\ \{e\}) : t_x \to t_r} \qquad \frac{\Gamma \vdash e_1 : t_2 \to t \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1(e_2) : t}$$

the evaluation of a well-typed program will be guaranteed not to encounter a certain class of errors. Note that a well-typed program may still encounter errors in another class: non-termination and division by zero in our small language. Also note that a program that fails to typecheck according to our rules may still evaluate without any errors. (For example **if** true **then** 1 **else** $+(2, \mathsf{false})$ does not typecheck but evaluates to 1.) In summary, the type rules are a static approximation of what constitutes "good behavior." The fact that such static approximations can never be exact means that one can always develop a more sophisticated type system that accepts a different class of programs.

Before getting to the type rules, note that the syntax forces every variable declaration to be associated with a type: the formal parameter of a function must be given a type, and the function declaration itself must be given a return type. Intuitively speaking the process of type checking is to make sure that every use of a variable is consistent with its declaration. To facilitate this process, variable declarations and their types are collected in a table (called an environment and denoted with the letter Γ) which is propagated by the type rules following the usual scoping rules. The judgment $\Gamma \vdash e : t$ means that given the environment Γ, we can prove that expression e has type t. Each rule proves a judgment for a certain kind of expression making assumptions about the judgments for the subexpressions.

Here is an example derivation for the factorial example:

$$\frac{\dfrac{\dfrac{\dfrac{}{x : \texttt{int}, f : \texttt{int} \to \texttt{int} \vdash x : \texttt{int}} \quad \dfrac{}{x : \texttt{int}, f : \texttt{int} \to \texttt{int} \vdash 0 : \texttt{int}}}{x : \texttt{int}, f : \texttt{int} \to \texttt{int} \vdash = (x, 0) : \texttt{bool}} \quad \dfrac{}{x : \texttt{int}, f : \texttt{int} \to \texttt{int} \vdash 1 : \texttt{int}} \quad C}{x : \texttt{int}, f : \texttt{int} \to \texttt{int} \vdash \textbf{if } = (x, 0) \textbf{ then } 1 \textbf{ else } *(x, f(-(x, 1))) : \texttt{int}}}{\emptyset \vdash (\textbf{fun}\ \texttt{int}\ f\ (\texttt{int}\ x)\ \{\textbf{if } = (x, 0) \textbf{ then } 1 \textbf{ else } *(x, f(-(x, 1)))\}) : \texttt{int} \to \texttt{int}}$$

where the derivation C is:

$$\frac{\dfrac{}{x : \texttt{int}, f : \texttt{int} \to \texttt{int} \vdash x : \texttt{int}} \quad \dfrac{\dfrac{}{x : \texttt{int}, f : \texttt{int} \to \texttt{int} \vdash f : \texttt{int} \to \texttt{int}} \quad \dfrac{\dfrac{}{x : \texttt{int}, f : \texttt{int} \to \texttt{int} \vdash x : \texttt{int}} \quad \dfrac{}{x : \texttt{int}, f : \texttt{int} \to \texttt{int} \vdash 1 : \texttt{int}}}{x : \texttt{int}, f : \texttt{int} \to \texttt{int} \vdash -(x, 1) : \texttt{int}}}{x : \texttt{int}, f : \texttt{int} \to \texttt{int} \vdash f(-(x, 1)) : \texttt{int}}}{x : \texttt{int}, f : \texttt{int} \to \texttt{int} \vdash *(x, f(-(x, 1))) : \texttt{int}}$$

4 Dynamic Semantics

The dynamic semantics is a function that maps a syntactically valid program to a value. The function is partial since programs may diverge, cause errors like division by zero, or get stuck if a nonsensical operation such as adding 1 to true is attempted at runtime. In the next section we will be concerned with the proof that well-typed programs can never get stuck during evaluation.

To be close to the ASM framework, the dynamic semantics is specified using an abstract machine. The machine has three components: the code being evaluated, the environment that holds the values for the free variables in the code, and the stack of activation records. The evaluation proceeds in steps: at each step, a subexpression of the entire program is chosen for evaluation. The subexpressions of the program are gradually replaced by their values until the entire program reduces to a value or evaluation gets stuck.

To specify this evaluation formally, we first need to define the set of values and then extend the syntax of expressions to accommodate the fact that some subexpressions may be replaced by values or runtime errors.

Values	v	$::=$	$\underline{n}$	*integer values*
		$\mid$	$\underline{\textsf{true}} \mid \underline{\textsf{false}}$	*boolean values*
		$\mid$	$\langle \underline{\mathit{clos}}(\textbf{fun}\ t\ f\ (t\ x)\ \{e\}), \rho \rangle$	*function values (closures)*
		$\mid$	$\underline{\mathit{DivZero}}$	*error condition for division by zero*
Environments	ρ	$::=$	$\{x_1 = v_1, x_2 = v_2, \cdots, x_n = v_n\}$	
(Runtime) Expressions	e	$::=$	n	*integer constants*
		$\mid$	x	*variables*
		$\mid$	$o(e, e)$	*primitive applications*
		$\mid$	$\textsf{true} \mid \textsf{false}$	*boolean constants*
		$\mid$	$\textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e$	*conditional expressions*
		$\mid$	$(\textbf{fun}\ t\ f\ (t\ x)\ \{e\})$	*user-defined recursive functions*
		$\mid$	$e(e)$	*function applications*
		$\mid$	$\underline{n}$	*integer values*
		$\mid$	$\underline{\textsf{true}} \mid \underline{\textsf{false}}$	*boolean values*
		$\mid$	$\langle \underline{\mathit{clos}}(\textbf{fun}\ t\ f\ (t\ x)\ \{e\}), \rho \rangle$	*function values (closures)*
		$\mid$	$\underline{\mathit{DivZero}}$	*error condition for division by zero*

The process of choosing one subexpression to evaluate is best explained using the notion of evaluation contexts defined below.

Evaluation contexts	E	$::=$	$[\,]$	*empty context*
		$\mid$	$o(E, e)$	*evaluate left argument first*
		$\mid$	$o(v, E)$	*when done with left argument, go to right*
		$\mid$	$\textbf{if}\ E\ \textbf{then}\ e\ \textbf{else}\ e$	*need the value of the test*
		$\mid$	$E(e)$	*left first*
		$\mid$	$v(E)$	*then right*

It is easy to verify that every expression e has a *unique decomposition* into an evaluation context E and a subexpression of interest. This is formalized in the following lemma.

Lemma 4.1 (Unique Decomposition) *Every expression e is in one (and only one) of the following forms: one of the following:*

- *a value v,*
- *an evaluation context E filled with:*
  - *an integer constant n,*
  - *a variable x,*
  - *a boolean constant true or false,*
  - *a function declaration $(\textbf{fun}\ t\ f\ (t\ x)\ \{e\})$,*
  - *an application of a primitive operation to two values $o(v_1, v_2)$,*

- *a conditional expression with an evaluated test position* ***if*** v ***then*** e_1 ***else*** e_2,
- *an application of two values* $v_1(v_2)$.

Each activation record is of the form (E, ρ). In other words, when a function call occurs within an evaluation context E, we save the evaluation context on the stack together with the environment needed for its free variables.

To evaluate a program e, the abstract machine is put in the initial state $\langle e, \emptyset, []\rangle$. A successful evaluation terminates with a state of the form $\langle v, \rho, []\rangle$ for some value v and environment ρ. The transitions of the machine are:

$$\begin{array}{rcl}
\langle v, \rho', (E,\rho) : \kappa\rangle & \longmapsto & \langle E[v], \rho, \kappa\rangle \\
\\
\langle E[n], \rho, \kappa\rangle & \longmapsto & \langle E[\underline{n}], \rho, \kappa\rangle \\
\langle E[x], \rho, \kappa\rangle & \longmapsto & \langle E[\rho(x)], \rho, \kappa\rangle \\
\langle E[\mathsf{true}], \rho, \kappa\rangle & \longmapsto & \langle E[\underline{\mathsf{true}}], \rho, \kappa\rangle \\
\langle E[\mathsf{false}], \rho, \kappa\rangle & \longmapsto & \langle E[\underline{\mathsf{false}}], \rho, \kappa\rangle \\
\langle E[(\textbf{fun}\ t'\ f\ (t\ x)\ \{e\})], \rho, \kappa\rangle & \longmapsto & \langle E[\langle \underline{\mathit{clos}}(\textbf{fun}\ t'\ f\ (t\ x)\ \{e\}), \rho\rangle], \rho, \kappa\rangle \\
\\
\langle E[+(\underline{n_1}, \underline{n_2})], \rho, \kappa\rangle & \longmapsto & \langle E[\underline{n_1 + n_2}], \rho, \kappa\rangle \\
\langle E[*(\underline{n_1}, \underline{n_2})], \rho, \kappa\rangle & \longmapsto & \langle E[\underline{n_1 * n_2}], \rho, \kappa\rangle \\
\langle E[-(\underline{n_1}, \underline{n_2})], \rho, \kappa\rangle & \longmapsto & \langle E[\underline{n_1 - n_2}], \rho, \kappa\rangle \\
\langle E[/(\underline{n_1}, \underline{n_2})], \rho, \kappa\rangle & \longmapsto & \langle E[\underline{n_1/n_2}], \rho, \kappa\rangle \text{ if } n_2 \neq 0 \\
\langle E[/(\underline{n}, \underline{0})], \rho, \kappa\rangle & \longmapsto & \langle E[\underline{\mathit{DivZero}}], \rho, \kappa\rangle \\
\langle E[=(\underline{n}, \underline{n})], \rho, \kappa\rangle & \longmapsto & \langle E[\underline{\mathsf{true}}], \rho, \kappa\rangle \\
\langle E[=(\underline{n_1}, \underline{n_2})], \rho, \kappa\rangle & \longmapsto & \langle E[\underline{\mathsf{false}}], \rho, \kappa\rangle \text{ if } n_1 \neq n_2 \\
\langle E[<(\underline{n_1}, \underline{n_2})], \rho, \kappa\rangle & \longmapsto & \langle E[\underline{\mathsf{true}}], \rho, \kappa\rangle \text{ if } n_1 < n_2 \\
\langle E[<(\underline{n_1}, \underline{n_2})], \rho, \kappa\rangle & \longmapsto & \langle E[\underline{\mathsf{false}}], \rho, \kappa\rangle \text{ if } n_1 \geq n_2 \\
\langle E[o(\underline{n}, \underline{\mathit{DivZero}})], \rho, \kappa\rangle & \longmapsto & \langle E[\underline{\mathit{DivZero}}], \rho, \kappa\rangle \\
\langle E[o(\underline{\mathit{DivZero}}, e)], \rho, \kappa\rangle & \longmapsto & \langle E[\underline{\mathit{DivZero}}], \rho, \kappa\rangle \\
\\
\langle E[\textbf{if}\ \underline{\mathsf{true}}\ \textbf{then}\ e_1\ \textbf{else}\ e_2], \rho, \kappa\rangle & \longmapsto & \langle E[e_1], \rho, \kappa\rangle \\
\langle E[\textbf{if}\ \underline{\mathsf{false}}\ \textbf{then}\ e_1\ \textbf{else}\ e_2], \rho, \kappa\rangle & \longmapsto & \langle E[e_2], \rho, \kappa\rangle \\
\langle E[\textbf{if}\ \underline{\mathit{DivZero}}\ \textbf{then}\ e_1\ \textbf{else}\ e_2], \rho, \kappa\rangle & \longmapsto & \langle E[\underline{\mathit{DivZero}}], \rho, \kappa\rangle \\
\\
\langle E[\mathit{cl}(v)], \rho, \kappa\rangle & \longmapsto & \langle e, \rho'[x := v, f := \mathit{cl}], (E,\rho) : \kappa\rangle \\
 & & \text{where } \mathit{cl} = \langle \underline{\mathit{clos}}(\textbf{fun}\ t'\ f\ (t\ x)\ \{e\}), \rho'\rangle \\
 & & \text{and } v \neq \underline{\mathit{DivZero}} \\
\langle E[\underline{\mathit{DivZero}}(v)], \rho, \kappa\rangle & \longmapsto & \langle E[\underline{\mathit{DivZero}}], \rho, \kappa\rangle \\
\langle E[\langle \underline{\mathit{clos}}(\textbf{fun}\ t'\ f\ (t\ x)\ \{e\}), \rho'\rangle(\underline{\mathit{DivZero}})], \rho, \kappa\rangle & \longmapsto & \langle E[\underline{\mathit{DivZero}}], \rho, \kappa\rangle
\end{array}$$

5 Type Safety

Intuitively type safety means that if a program typechecks then its evaluation cannot get stuck. In other words, if we are given a machine state with a well-typed program, then there must be a way to make progress and the resulting state must also consist of a well-typed program. The only exception is of course when we reach a final state of the form $\langle v, \rho, []\rangle$ where the program has been completely evaluated.

We will talk about the proof in class. Writing this proof in as much as detail as you can is your homework.

References

[1] HARPER, R. *Programming Languages: Theory and Practice.* Draft, 2001.

[2] STÄRK, R. F., SCHMID, J., AND BÖRGER, E. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer-Verlag, 2001.

A AsmGofer Implementation

```
{--

File: minML.gs

A small typed functional language and its formalization in AsmGofer...

by Amr Sabry (based on Ch. 3-5 of Programming Languages: Theory and
   Practice, Robert Harper, Draft of Dec. 2000 and using the style
   developed for the JBook)

--}

--------------------------------------------------------------------------------
-- Annotated abstract syntax trees:
--  * initially the AST has no types and only uses syntactic constructors
--  * after typechecking the nodes are decorated with types
--  * during evaluation syntactic constructors are replaced by semantic values
--------------------------------------------------------------------------------

data MLType = T_Int | T_Bool | T_Arrow MLType MLType
instance AsmTerm MLType

data MLOp = O_Plus | O_Times | O_Minus | O_Div | O_Equal | O_LessThan
instance AsmTerm MLOp

data MLTerm a = Term a [MLTerm a]
instance AsmTerm a => AsmTerm (MLTerm a)

data MLForm =                             -- a
  -- Syntactic forms
    S_Num Int                             -- []
  | S_Var String                          -- []
  | S_Prim MLOp                           -- [MLTerm a, MLTerm a]
  | S_True                                -- []
  | S_False                               -- []
  | S_If                                  -- [MLTerm a, MLTerm a, MLTerm a]
  | S_Fun MLType String (String,MLType)   -- [MLTerm a]
  | S_App                                 -- [MLTerm a, MLTerm a]
  -- Runtime values or errors
  | R_Val MLValue                         -- []
  | R_DivZero                             -- []
instance AsmTerm MLForm

type MLExp = MLTerm MLForm
type TypedMLExp = MLTerm (MLForm,MLType)

data MLValue = V_Num Int | V_True | V_False | V_Closure MLExp VEnv
instance AsmTerm MLValue
```

```
type VEnv = [(String,MLValue)]

------------------------------------------------------------------------------
-- Static semantics:
--   * typechecking takes an AST and returns another AST with all the types
--     or aborts if there is an error
-------------------------------------------------------------------------------

type TEnv = [(String,MLType)]

tlookup :: String -> TEnv -> MLType
tlookup v [] = error ("Typechecking: unbound variable " ++ v)
tlookup v ((s,t):r) = if v == s then t else tlookup v r

typeOf :: TypedMLExp -> MLType
typeOf (Term (_,t) _) = t

typecheck :: MLExp -> TEnv -> TypedMLExp
typecheck exp tenv =
  case exp of

    Term (S_Num i) [] -> Term (S_Num i, T_Int) []

    Term (S_Var v) [] -> Term (S_Var v, tlookup v tenv) []

    Term (S_Prim b) [e1,e2] | b == O_Equal || b == O_LessThan ->
      let e1’ = typecheck e1 tenv
          e2’ = typecheck e2 tenv
          rt = case (typeOf e1’, typeOf e2’) of
                 (T_Int,T_Int) -> T_Bool
                 (t1,t2) ->
                   error ("Typechecking: operator <"
                   ++ show b ++ "> requires operands of type int; found <"
                   ++ show t1 ++ "> and <" ++ show t2 ++ ">")
      in Term (S_Prim b, rt) [e1’,e2’]

    Term (S_Prim b) [e1,e2] | b == O_Plus || b == O_Minus ||
                              b == O_Times || b == O_Div ->
      let e1’ = typecheck e1 tenv
          e2’ = typecheck e2 tenv
          rt = case (typeOf e1’, typeOf e2’) of
                 (T_Int,T_Int) -> T_Int
                 (t1,t2) ->
                   error ("Typechecking: operator <"
                   ++ show b ++ "> requires operands of type int; found <"
                   ++ show t1 ++ "> and <" ++ show t2 ++ ">")
      in Term (S_Prim b, rt) [e1’,e2’]

    Term S_True [] -> Term (S_True, T_Bool) []

    Term S_False [] -> Term (S_False, T_Bool) []

    Term S_If [e1,e2,e3] ->
      let e1’ = typecheck e1 tenv
          e2’ = typecheck e2 tenv
          e3’ = typecheck e3 tenv
          rt = if typeOf e1’ == T_Bool && typeOf e2’ == typeOf e3’
               then typeOf e2’
               else error ("Typechecking: if requires a boolean and "
               ++ "two expressions of the same type; found <"
               ++ show (typeOf e1’) ++ ">, <"
               ++ show (typeOf e2’) ++ ">, and <"
               ++ show (typeOf e3’) ++ ">")
      in Term (S_If,rt) [e1’,e2’,e3’]

    Term (S_Fun rt fn (pn,pt)) [e] ->
      let tenv’ = (fn, T_Arrow pt rt) : (pn, pt) : tenv
          e’ = typecheck e tenv’
          ct = if typeOf e’ == rt
               then T_Arrow pt rt
               else error ("Typechecking: declared return type <" ++ show rt
                           ++ "> does not agree with actual return type <"
                           ++ show (typeOf e’) ++ ">")
      in Term (S_Fun rt fn (pn,pt), ct) [e’]

    Term S_App [e1,e2] ->
      let e1’ = typecheck e1 tenv
          e2’ = typecheck e2 tenv
          ct = case (typeOf e1’, typeOf e2’) of
                 (T_Arrow t2 t, t2’) | t2 == t2’ -> t
                 (t1,t2) ->
                   error ("Typechecking: attempting to apply a "
                   ++ "function of type <"
                   ++ show t1 ++ "> to an argument of type <"
                   ++ show t2 ++ ">")
      in Term (S_App,ct) [e1’,e2’]

    exp -> error ("Typecheck: unexpected expression " ++ show exp)

--------------------------------------------------------------------------------
-- Positions
--------------------------------------------------------------------------------

type Pos = [Int]

up :: Pos -> Pos
up [] = [] -- NOT an error (used to return from top level evaluation)
up ds = init ds

firstPos :: Pos
firstPos  = []

down :: (Pos,Int) -> Pos
down (ds,d) = ds ++ [d]

-- During evaluation we replace syntactic constructors by dynamic values or
-- runtime errors
substMLExp :: (MLExp, MLExp, Pos) -> MLExp
substMLExp (e, (Term _ _), []) = e
substMLExp (e, (Term a ts), p:ps) =
  let (lts,rt:rts) = splitAt p ts
  in Term a (lts ++ [ substMLExp(e,rt,ps) ] ++ rts)
substMLExp (e1, e2, p) =
  error ("substMLExp: unexpected arguments "
         ++ show e1 ++ ", "
         ++ show e2 ++ ", and "
         ++ show p)

-- context takes an expression and a position and returns
-- the subexpression at the position
context :: (MLExp, Pos) -> MLExp
context (e,[]) = e
context (Term _ es, i:is) = context (es!!i, is)
context (e,p) = error ("context: unexpected expression and position "
                        ++ show e ++ " and "
                        ++ show p)

-------------------------------------------------------------------------------
-- ASM states
-------------------------------------------------------------------------------

code :: Dynamic MLExp
code = initVal "code" asmDefault

pos :: Dynamic Pos
pos = initVal "pos" firstPos

env :: Dynamic VEnv
env = initVal "env" []

type SFrame = (MLExp,Pos,VEnv)

stack :: Dynamic [SFrame]
stack = initVal "stack" []

initialize :: MLExp -> IO ()
initialize e =
  fire1 (do code := e
            pos := firstPos
            env := []
            stack := [])

-------------------------------------------------------------------------------
-- Evaluation:
--  * proceeds by finding a position where we can evaluate
--  * performs the evaluation
--  * replaces the constructor at the position with the value
-------------------------------------------------------------------------------

vlookup :: String -> VEnv -> MLValue
vlookup v [] = error ("vlookup: unexpected unbound variable" ++ v)
vlookup v ((s,va):r) = if v == s then va else vlookup v r

execML :: Rule ()
execML =
  case context (code,pos) of

    Term (S_Num i) [] -> yield (Term (R_Val (V_Num i)) [])

    Term (S_Var v) [] -> if v ‘elem‘ map fst env
                         then yield (Term (R_Val (vlookup v env)) [])
                         else skip

    Term (S_Prim op) [Term (R_Val v1) [], Term (R_Val v2) []] ->
      if op == O_Div && v2 == V_Num 0
      then yield (Term R_DivZero [])
      else yield (Term (applyOp op v1 v2) [])

    Term (S_Prim op) [Term (R_Val v1) [], Term R_DivZero []] ->
      yield (Term R_DivZero [])

    Term (S_Prim op) [Term (R_Val v1) [], e2] -> pos := down (pos,1)

    Term (S_Prim op) [Term R_DivZero [], e2] -> yield (Term R_DivZero [])

    Term (S_Prim op) [e1,e2] -> pos := down (pos,0)

    Term S_True [] -> yield (Term (R_Val V_True) [])

    Term S_False [] -> yield (Term (R_Val V_False) [])

    Term (S_If) [Term R_DivZero [], e2, e3] -> yield (Term R_DivZero [])

    Term (S_If) [Term (R_Val v1) [], e2, e3] ->
      case v1 of
        V_True -> yield e2
        V_False -> yield e3
        _ -> skip

    Term (S_If) [e1,e2,e3] -> pos := down (pos,0)

    Term (S_Fun rt fn (pn,pt)) [e] ->
      yield (Term (R_Val (V_Closure (Term (S_Fun rt fn (pn,pt)) [e]) env)) [])

    Term S_App [Term (R_Val v1) [], Term R_DivZero []] -> yield (Term R_DivZero [])

    Term S_App [Term (R_Val v1) [], Term (R_Val v2) []] ->
      case v1 of
        V_Closure (Term (S_Fun rt fn (pn,pt)) [e]) lenv ->
          do stack := (code,pos,env) : stack
             code := e
             pos := firstPos
             env := (pn,v2) : (fn,v1) : lenv
        _ -> skip

    Term S_App [Term R_DivZero [], e2] -> yield (Term R_DivZero [])

    Term S_App [Term (R_Val v) [], e2] -> pos := down (pos,1)

    Term S_App [e1,e2] -> pos := down (pos,0)
```

```
    Term (R_Val v) ts ->
      case stack of
        [] -> skip
        (c1,p1,e1):s1 -> do code := substMLExp (Term (R_Val v) ts, c1, p1)
                            pos := up p1
                            env := e1
                            stack := s1

    Term R_DivZero [] ->
      case stack of
        [] -> skip
        (c1,p1,e1) : s1 -> do code := substMLExp (Term R_DivZero [], c1, p1)
                              pos := up p1
                              env := e1
                              stack := s1

    e -> error ("execML: unexpected expression " ++ show e)

yield :: MLExp -> Rule ()
yield result = do
   code := substMLExp (result , code , pos)
   pos := up pos

--------------------------------------------------------------------------------
-- Evaluation
--------------------------------------------------------------------------------

eval :: MLExp -> IO ()
eval e = do putStr "-----------------------------------------\nTypechecking..."
            putStr "\nExpression: "
            print e
            putStr "has type: "
            print (typeOf (typecheck e []))
            putStr "-----------------------------------------\nEvaluating...\n"
            initialize e
            fixpoint (trace printState execML)
            printValue

applyOp :: MLOp -> MLValue -> MLValue -> MLForm
applyOp O_Plus (V_Num i1) (V_Num i2) = R_Val (V_Num (i1+i2))
applyOp O_Times (V_Num i1) (V_Num i2) = R_Val (V_Num (i1*i2))
applyOp O_Minus (V_Num i1) (V_Num i2) = R_Val (V_Num (i1-i2))
applyOp O_Div (V_Num i1) (V_Num i2) = R_Val (V_Num (i1/i2))
applyOp O_Equal (V_Num i1) (V_Num i2) =
  R_Val (if i1 == i2 then V_True else V_False)
applyOp O_LessThan (V_Num i1) (V_Num i2) =
  R_Val (if i1 < i2 then V_True else V_False)
applyOp op v1 v2 =
  error ("applyOp: unexpected operator and values"
  ++ show op ++ ", " ++ show v1 ++ ", and " ++ show v2)

printState :: IO ()
printState = do putStr "<code = "
                print code
                putStr ",pos  = "
                print pos
                putStr ",env  = "
                print env
                putStr ",stack size = "
                print (length stack)
                putStr ">\n--------------------\n"

printValue :: IO ()
printValue = do putStr "VALUE = "
                case code of
                  Term (R_Val v) [] -> print v
                  Term R_DivZero [] -> putStr "Exception: division by zero"
                  _ -> error ("Unexpected value"  ++ show code)

--------------------------------------------------------------------------------
-- Examples
--------------------------------------------------------------------------------

numE e = Term (S_Num e) []
varE e = Term (S_Var e) []
trueE = Term S_True []
falseE = Term S_False []
primE b e1 e2 = Term (S_Prim b) [e1,e2]
addE e1 e2 = primE O_Plus e1 e2
divE e1 e2 = primE O_Div e1 e2
lessE e1 e2 = primE O_LessThan e1 e2
ifE e1 e2 e3 = Term S_If [e1,e2,e3]
funE rt fn (pn,pt) b = Term (S_Fun rt fn (pn,pt)) [ b ]
appE e1 e2 = Term S_App [e1,e2]

t1 = addE (addE (numE 1) (numE 2)) (addE (numE 3) (numE 4))
t2 = addE (addE (numE 1) (numE 2)) (addE (divE (numE 1) (numE 0)) (numE 4))
t3 = appE (funE T_Int "f" ("x",T_Int) (appE (varE "f") (varE "x"))) (numE 0)
t4 = ifE trueE (numE 1) t3
t5 = funE T_Int "f" ("x",T_Int) (ifE (primE O_Equal (varE "x") (numE 0))
                                     (numE 1)
                                     (primE O_Times (varE "x")
                                        (appE (varE "f")
                                              (primE O_Minus (varE "x") (numE 1)))))
t6 = appE t5 (numE 5) -- factorial of 5
t7 = funE T_Int "f" ("x",T_Int) (ifE (primE O_Equal (varE "x") (numE 0))
                                     (numE 1)
                                     (primE O_Times
                                        (appE (varE "f")
                                              (primE O_Minus (varE "x") (numE 1)))
                                        (varE "x")))
t8 = appE t7 (numE 5) -- checking environment after popping stack

t9 = divE (numE 1) (numE 0)
t10 = lessE t9 (numE 0) -- a boolean DivZero
t11 = ifE t10 t5 t5 -- an int->int divZero

t12 = appE t11 (numE 0)
```

```
--------------------------------------------------------------------------------
-- Printing
--------------------------------------------------------------------------------

showSepBy :: String -> [ShowS] -> ShowS
showSepBy _ []       = id
showSepBy _ [x]      = x
showSepBy sep (x:xs) = x . showString sep . showSepBy sep xs

instance Text MLType where
  showsPrec _ T_Int = showString "int"
  showsPrec _ T_Bool = showString "bool"
  showsPrec _ (T_Arrow t1 t2) = shows t1 . showString " -> " . shows t2
  showsPrec _ t = error ("show: unexpected type " ++ show t)

instance Text MLOp where
  showsPrec _ O_Plus = showString "+"
  showsPrec _ O_Times = showString "*"
  showsPrec _ O_Minus = showString "-"
  showsPrec _ O_Div = showString "/"
  showsPrec _ O_Equal = showString "=="
  showsPrec _ O_LessThan = showString "<"
  showsPrec _ op = error ("show: unexpected operator " ++ show op)

instance Text a => Text (MLTerm a) where
  showsPrec _ (Term e []) = shows e
  showsPrec _ (Term e es) =
    shows e . showString "{" . showSepBy " , " (map shows es) . showString "}"

instance Text MLForm where
  showsPrec _ (S_Num i) = shows i
  showsPrec _ (S_Var v) = showString v
  showsPrec _ (S_Prim bop) = shows bop
  showsPrec _ S_True = showString "true"
  showsPrec _ S_False = showString "false"
  showsPrec _ S_If = showString "if"
  showsPrec _ (S_Fun rt fn (pn,pt)) =
    shows rt .
    showString (" " ++ fn ++ " (" ++ pn) .
    showString ":" .
    shows pt .
    showString ") "
  showsPrec _ S_App = showString "@"
  showsPrec _ (R_Val v) = shows v
  showsPrec _ R_DivZero = showString "DivZero"
  showsPrec _ c  = error ("show: unexpected form " ++ show c)

instance Text MLValue where
  showsPrec _ (V_Num i) = shows i
  showsPrec _ V_True = showString "true"
  showsPrec _ V_False = showString "false"
  showsPrec _ (V_Closure exp env) = showString "<closure>"
  showsPrec _ v = error ("show: unexpected value " ++ show v)
```